**IBM** Systems Reference Library

# IBM 7040/7044 Operating System (16/32K)

## Debugging Facilities

This publication describes the 7040/7044 (16/32K) Operating
System Debugging Package and the dump routines available with
the 7040/7044 Operating System, Version 9. The Debugging
Package is a programming aid that enables the user to obtain
dynamic dumps of specified areas of core storage and machine
registers during program execution.

Among the subjects discussed in this publication are Compile-
Time Debugging for COBOL programs, Load-Time Debugging for
FORTRAN and MAP programs, Dump Routines and Parameters,
and the Snapshot Routine.

# Preface

The IBM 7040/7044 Operating System Debugging Package provides a means of taking highly selective dumps of core storage areas and machine registers with a minimum of programming effort. By carefully selecting the areas to be dumped and the time at which to dump them, the user can obtain information valuable in locating and correcting program errors. The facilities described in this publication pertain to FORTRAN IV, COBOL, and MAP program debugging.

As a prerequisite to understanding this publication, the reader should be familiar with the IBJOB Processor, as described in the IBM publications *IBM 7040/7044 Operating System (16/32K): Programmer's Guide,* and *IBM 7040/7044 Operating System (16/32K): Operator's Guide.* He should also be familiar with at least one of the programming languages accepted by the processor that are described in the following IBM publications:

> *IBM 7040/7044 Operating System (16/32K): Macro Assembly Program (MAP) Language,* Form C28-6335.
>
> *IBM 7040/7044 Operating System (16/32): FORTRAN IV Language,* Form C28-6329.
>
> *IBM 7040/7044 Operating System (16/32K): COBOL Language,* Form C28-6336.

The actual debugging languages themselves are based on COBOL (for COBOL programs) and FORTRAN IV (for FORTRAN IV and MAP programs). The MAP programmer who is totally unfamiliar with FORTRAN should be able to use all the facilities described in this publication with limited reference to the FORTRAN language publication listed above.

The 7040/7044 (16/32K) Processor Debugging Package requires the same minimum machine configuration as the 7040/7044 IBJOB Processor. The only difference lies in the use of the load-time facility, where the Debug Work Unit or the system checkpoint unit is required for the intermediate debugging output.

IBM 7040/7044 Operating System (16/32K)
Debugging Facilities
Addenda and Errata to Form C28-6803-1

Attached are replacement pages for the publication <u>IBM 7040/7044 Operating System</u> <u>(16/32K), Debugging Facilities,</u> Form C28-6803-1. The numbers of the amended pages are 6, 7, 9, 11, 13, 16, 27, and 28.

Revisions to the text are indicated by a vertical line to the left of the change.

File this Newsletter at the back of the publication. It will provide a reference to changes, a method of determining that all amendments have been received, and a check for determining if the publication contains the proper pages.

# Contents

The problem of locating program errors rapidly and efficiently is of major concern to all computer users. The 7040/7044 (16/32K) Processor has been extended to include a debugging package as an aid in locating and correcting errors. To diagnose program errors, the programmer may wish to obtain information at key points in his program. The debugging package enables him to manipulate data, control processing, and print out the contents of program areas or machine registers. To use the debugging package, the programmer writes a *debug request* in the appropriate debugging language. Each request specifies what action to take and when to take such action.

The debugging package provides two types of debugging: compile-time and load-time. Compile-time debugging is included with IBCBC at compilation to specify dumps at various points in a COBOL source program. The text of the debug requests is similar to the COBOL language. Load-time debugging uses the capabilities of IBMAP and IBLDR to provide debugging during the execution of a FORTRAN IV or MAP source program without recompiling or reassembling the program. The text of these debug requests is in a form similar to that of the FORTRAN IV language.

In addition to the debugging package, the 7040/7044 Operating System provides a dump program that can be used by object programs, system programs, or machine operators. This program lists the operator console panel, certain symbolic unit information, and specified portions of internal and external storage. It is not intended to replace object program symbolic debugging tools.

## Notation Conventions

The following conventions apply to all card formats given in this publication:

1. Brackets, [ ], indicate that the enclosed material may be omitted.

2. Braces, ( ), indicate that the user must make a choice of the enclosed material.

3. Upper-case words, if used, must be present in the form indicated.

4. Lower-case words represent generic quantities whose values must be supplied by the user.

The statement formats for COBOL and FORTRAN IV adhere to the notation conventions given for each in their respective language publications.

# Compile-Time Debugging for COBOL Programs

The compile-time facility of the debugging package enables the COBOL programmer to include debug requests with his source-language program. Debug requests are compiled with the source program and are executed at object time. The text of the requests is similar to the procedural text of COBOL. In addition, a special count-conditional statement is provided. Since the procedural capabilities of the COBOL compiler are available, a user can be highly selective in specifying what is to be dumped. He can manipulate and test the values of intermediate results in his program and dump only pertinent and meaningful information without affecting execution of the program itself.

It is possible to delete the debug requests at load time without recompiling. The COBOL programmer may, if desired, take advantage of the load-time facility, including the Additional Load-Time Debugging Features, and debug from an assembly listing of his program.

## Compile-Time Debugging Packet

All compile-time requests for a given program are grouped together into a debugging packet and placed immediately after the $CBEND card of the associated source program.

## Compile-Time Debug Requests

Each compile-time debug request is headed by a $IBDBC control card. The $IBDBC card identifies individual requests and defines the point at which the request is to be executed. The $IBDBC card may contain any blanks desired for legibility except in a character string that is to be treated as a single parameter. The general form of this card is:

| 1 | 8 | 16-72 |
|---|---|---|
| $IBDBC | name | location [, FATAL] [, DUMP= symbolic unit] [, MARKER= file-name] |

The parameters in the card are described as:

name
   An optional user-assigned control-section name. Use of this parameter makes it possible to delete the request at load time. The name must be a unique control-section name consisting of at most six alphabetic or numeric characters. At least one of the characters must be alphabetic.

location
   The COBOL section-name or paragraph-name (qualified, if necessary) indicating the point in the program at which the request is to be executed. Debug request statements are executed as if they were physically placed in the source programs following the section-name or paragraph-name specified, but preceding the text associated with that name.

FATAL
   If FATAL is specified, severity codes normally assigned to

errors in COBOL statements will also be generated for errors in debug request statements. If FATAL is not specified and in the procedural text of a debug request an error is encountered that would normally be given a severity code of 2 or greater, the error message will be given a severity code of 1. An attempt will be made to interpret the statement; loading and execution of the object program will not be prevented. If interpretation is impossible, the erroneous statement will be discarded (but not the entire request, if it consists of more than one statement).
   Note that an error in *location* on an $IBDBC card or on an undefined symbol in the procedural statement of a debug request is always identified as a level 2 error, regardless of whether FATAL is specified.

DUMP=symbolic unit
   This option indicates the unit on which the debugging output is to be written. The *symbolic unit* can be the system output unit, OU; or a utility unit, Uxx (where xx is 01, 02, . . . 20). When this option is not specified, the debugging output will be written on the system output unit. The unit specified in the DUMP option must not be the same as an output unit used in the COBOL program.

MARKER=file-name
   This option causes a dump indication record to be written on the output file specified by *file-name* each time a debugging display occurs at the location specified in this $IBDBC card. *file-name* is the name used in the FD entry to name the file.

Dump indication records have the following format:
***** DUMP NO. xxxxx WRITTEN ON u, c, nn *****
where:

| xxxxx | is n when the nth debugging display is being executed. |
| u | is the output medium designation such as T for tape or M for disk/drum modules. |
| c | is the channel to which the output medium is attached. |
| nn | is the unit number. |

When the above options cannot fit between columns 8 and 72 of the $IBDBC card, this control card can be extended if the remaining options are inserted between columns 8 and 72 of subsequent cards. These extension cards must contain a hyphen in column 7, and columns 1 through 6 must be blank.

Note that the above options must appear on the $IBDBC card in the order in which they are described.

The text of the debug request follows immediately after the $IBDBC card. The text may consist of any valid procedural statements conforming to the requirements of the COBOL language and format and the count conditional statement described in the following text.

A compile-time debug request is terminated by another $IBDBC card or any other control card with a $ in column one.

## DISPLAY Verb

When used for debugging, the COBOL verb DISPLAY is modified to write on the symbolic unit specified by the DUMP parameter on the $IBDBC card or on the system output tape.

## Count-Conditional Statement

A count-conditional statement available only for use in debug requests allows the programmer to qualify the time at which a debugging action should be taken. The count-conditional statement has the same structure as the IF statement in COBOL (conditional, true option, false option) and may be used in the same manner; i.e., it may be nested within other count-conditional statements or IF statements and may have other count-conditional statements or IF statements nested within it. The general form of the count-conditional statement is:

ON k [AND EVERY m] [UNTIL n] statement-1

$\left[ \begin{Bmatrix} \text{ELSE} \\ \text{OTHERWISE} \end{Bmatrix} \text{statement-2} \right]$

The letters $k$, $m$, and $n$ are positive integers.

If AND EVERY $m$ is not specified, but UNTIL $n$ is specified, $m$ is assumed to be 1. The UNTIL option means up to but not including the $n$th time. If neither AND EVERY $m$ nor UNTIL $n$ is specified, action will take place only the $k$th time.

EXAMPLES

ON 3 DISPLAY A.
> The third time the count-conditional statement is executed, A is displayed. No action is taken at any other time.

ON 4 UNTIL 8 DISPLAY A.
> A is displayed on the fourth, fifth, sixth, and seventh times through the count-conditional statement. No action is taken at any other time. (This example implies, and has the same effect as, the statement ON 4 AND EVERY 1 UNTIL 8 DISPLAY A.)

ON 5 AND EVERY 3 UNTIL 12 DISPLAY A.
> A is displayed on the fifth, eighth, and eleventh times through the count-conditional statement. No action is taken at any other time.

ON 3 AND EVERY 2 DISPLAY A.
> A is displayed on the third, fifth, seventh, ninth, . . . times through the statement. On the first, second, fourth, sixth, . . . times, no action is taken.

ON 2 AND EVERY 2 UNTIL 10 DISPLAY A ELSE DISPLAY B.
> A is displayed on the second, fourth, sixth, and eighth times through the statement. B is displayed at all other times.

## Compiler Limitations

In debugging statements, the word ON may not be used with the SIZE ERROR option as is normally permissible with arithmetic statements or with the DEPENDING option in the GO TO statement.

When the name associated with an unconditional GO TO is referred to by a symbolic debugging request, the transfer point specified by the GO TO statement should not be changed by an ALTER statement.

With a 16K Operating System whose nucleus can be reduced sufficiently, the user may include exponentiation in symbolic debug. However, if the nucleus cannot be reduced sufficiently, he will not be able to use exponentiation in his COBOL debug requests. (A 32K user can always include exponentiation in his COBOL debug requests.)

## Deleting Debug Requests

Any debug request may be deleted by using the $OMIT card. Its form is:

| 1 | 8 | 16-72 |
|---|---|-------|
| $OMIT | | name |

On the card, *name* is the control-section name of the request to be deleted. When used, $OMIT cards are placed after the $IBJOB card for the processor application and before the source deck or relocatable binary deck of the program involved.

## *Example of a Compile-Time Debugging Packet*

Figure 1 gives an example of a compile-time debugging packet. The numbers in the first line across indicate the card columns in which the various fields begin.

In the first request, on the first, fourth, and seventh times that control passes through point A in the program, Z is displayed (in its own format as defined in the source program) with the identifying heading, Z=.

In the second request, the value of T (with the identifying heading, T=) is displayed and its value replaces the value of S at the point in the program identified as B of C. Further, if S is unequal to T, S is also displayed. This request may be deleted at any time through the use of a $OMIT card with the following form:

| 1 | 8 | 16-72 |
|---|---|-------|
| $OMIT | | NAME |

Execution of the third request causes both the message, V OUT OF RANGE, V=, and the value of V to be displayed the first nine times that V is greater than VMAX when program control passes through point D. On the tenth time the request causes STOP RUN to be executed. The FATAL option on the $IBDBC card heading this request inhibits execution of the source program if the compiler encounters an error.

```
1       8   12  16

$IBDBC          A
                ON 1 AND EVERY 3 UNTIL 8 DISPLAY
                'Z='Z.
$IBDBC NAME     B OF C
                IF S NOT EQUAL TO T DISPLAY 'S='S.
                MOVE T TO S.   DISPLAY 'T='T.
$IBDBC          D, FATAL
                IF V GREATER THAN VMAX ON 1 UNTIL
                10 DISPLAY 'V OUT OF RANGE, V='V
                ELSE STOP RUN.
```
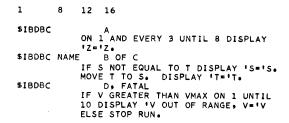
Figure 1. Example of a Compile-Time Debugging Packet

# Load-Time Debugging for FORTRAN IV and MAP Programs

The load-time facility of the debugging package enables FORTRAN IV and MAP programmers to include debug requests at load time to be executed with the object program. MAP object programs can be those generated by IBFTC and IBCBC as well as those written in MAP itself. The COBOL programmer may, if desired, take advantage of the load-time facility and debug from an assembly listing of his program. The COBOL programmer may use load-time debugging features while compile-time debug requests are still in his program. He may also use load-time debugging features at the same time he is using a SOMIT card to delete a debugging request. The FORTRAN IV programmer may also use the load-time facility at the MAP level.

The debugging package, which consists of three main parts, is another processor application under IBJOB. Following are the three parts:

1. The first is a preprocessor used to compile all debug requests and to associate these requests with programs to be debugged.

2. The second is an executable debugging program that is in core storage with the object program to be debugged and that performs necessary debugging.

3. The third is a postprocessor that processes all dumps produced during program execution and causes those dumps to be written in the proper format.

The debugging language used with the load-time facility is derived from the FORTRAN IV language. The statements available permit the programmer a high degree of flexibility in obtaining meaningful data. It is possible to perform arithmetic operations with object time values, to test results, and to freeze program action at a specified point and then dump selected information. In addition, the user can refer to symbols appearing in the source program by selecting the appropriate debugging dictionary option on the SIBFTC and SIBMAP control cards. (For additional information, see the section entitled "Debugging Dictionary.")

## Load-Time Debugging Packet

All load-time debug requests for a particular job run (any configuration of FORTRAN, COBOL, or MAP source and/or binary decks) are grouped into a debugging packet headed by a SIBDBL card and terminated by a SDEND card. The load-time debugging packet is placed in the job deck following any source decks. The debugging packet may be placed either before or after the binary decks of the job run. The packet may be followed only by the following cards: SENTRY, SLINK, SENDCH, SIBREL, and binary decks.

The program stacking option on the SIBJOB card must be SOURCE.

## $IBDBL Card

The general form of the SIBDBL card is:

| 1 | 8 | 16-72 |
|---|---|-------|
| $IBDBL | | [TRAP MAX=$n_1$] [,LINE MAX=$n_2$] [,Dump Marker Option] [,Debugging Work Unit Option] |

Columns 16-72, which are scanned in full, may contain any blanks desired for legibility, except in a character string that is to be treated as a single symbol or constant. Note that in the options TRAP MAX=$n_1$ and LINE MAX=$n_2$ the user may, if he desires, separate the equal sign with blanks.

The letters $n_1$ and $n_2$ are integers that should fall within the range 1 to 32767 decimal.

The contents of the variable field, which control the debugging output and may be used in any order are:

TRAP MAX=$n_1$

This specification causes termination of all debugging action after $n_1$ requests have been executed. If this option is omitted, the system assembly parameter for TRAP MAX, assembled as 30,000 decimal, will be used.

LINE MAX=$n_2$

The postprocessor will print no more than $n_2$ lines of output, excluding postmortem dumps. If this option is omitted, the system assembly parameter for LINE MAX, assembled as 1000 decimal, will be used.

### Dump Marker Option

$$\left[ \left\{ \begin{array}{l} \text{FORTRAN} \\ \text{JOBOU} \\ \text{JOBOUL} \\ \text{IOBS} = \text{file-name} \\ \text{IOOP} = \text{unit1} \end{array} \right\} \right]$$

This option specifies the IOCS level that the debugging routines will use to write dump markers each time a debug dump is taken. The IOCS level specified should be the same as that used by the object program so that the dump markers will be synchronized with object program output. These markers will match the dump numbers written with the debug dumps after the object program is executed. If the dump marker option is not taken, no marker will be written.

If FORTRAN is specified, the object program uses the FORTRAN input/output subroutines, and markers are written on FORTRAN logical unit 6 in BCD mode using the FORTRAN input/output subroutines.

If JOBOU or JOBOUL is specified, the object program uses the Output Editor.

If IOBS=*file-name* is specified, the object program uses the IOBS level of IOCS. The name associated with the output file is *file-name*. Markers will be written on this file according to the mode and record type defined in the file control block. Note that no marker will be written if the file is not open, or if the logical record length of a Type 1 file or the block size of any file is less than the necessary six words that indicate the length of the dump marker. Only one file may be specified for dump marking.

If IOOP=*unit1* is specified, the IOOP level of IOCS is used by the object program. *unit1* is a system output unit (OU), a system utility unit (U00-U99), or an intersystem reservation unit (I01-I20). Only one unit may be specified for dump marking.

### Debug Work Unit Option

[, DWU=unit2]

This option specifies the work unit for intermediate debugging output. *unit2* is any unit designation that would be valid on the $FILE card, except the following: NONE, *, LB1, LB2, IN, OU, PP, or card equipment. (For information on the $FILE card, see *IBM 7040/7044 Operating System (16/32K): Programmer's Guide*, Form C28-6318.) If this option is not taken, the debug work unit is assumed to be the system checkpoint unit (CK1). The assumed debug work unit is set by the $FILE card used when editing the debug preprocessor onto the system tape.

NOTE: Debug work unit and dump marker unit specifications are processed following the assignment of IBJOB work units, and prior to IBLDR processing of unit designations on $FILE cards and from FILE pseudo-operations. If the programmer wishes to assign the debug work unit to a specific unit, he should assign an intersystem reservation code to that unit on the IBSYS level or otherwise insure that IBJOB will not assign that unit as one of the system work units.

### $DEND Card

The $DEND card terminates the load-time debugging packet. The card format is:

| 1 | 8 | 16-72 |
|---|---|---|
| $DEND | | |

### Load-Time Debug Requests

Each load-time debug request is headed by a $DEBUG card, which identifies individual requests and specifies the point in the program at which the request is to be executed. There may be multiple requests per deck.

The general form of the $DEBUG card is:

| 1 | 8 | 16-72 |
|---|---|---|
| $DEBUG | deckname | location1 [, location2...] |

The variable field (columns 16-72) is scanned in full and, therefore, may contain any blanks desired for legibility, except in a character string that is to be treated as a single symbol or integer. The parameters of the $DEBUG card are:

deckname
> The name of the object deck to which this debug request applies. If this field is blank, the deck name specified on a preceding $DEBUG or $REDEF card will be assumed. If no deck name was specified, the request will be deleted.

Symbols in the request that follows are considered to be in this deck.

location1...
> The location(s) of the executable instruction(s) at which this debug request is to be inserted. A location may be specified in any of the following ways:
> 1. A statement number of an executable statement (FORTRAN IV only. See Appendix B, Restriction 6)
> 2. A symbol
> 3. A symbol ± an unsigned decimal integer
> 4. The characters =R followed by an unsigned octal integer for a relative location (i.e., relative to the load address of the deck)
> 5. The characters =A followed by an unsigned octal integer for an absolute location

The debug request will be executed as if it had been physically inserted in the deck at the location(s) specified. Debug request action occurs before the execution of the statement or instruction at the location. The locations at which debugging is specified will be modified to provide a transfer (STR instruction) to the debugging supervisor. Bits 1-11 will be set to zero and should not be changed by the object program. The transfer to the debugging supervisor is executed each time this location is reached, although conditional statements may cause action within the debug request to be skipped. The original instruction is always executed before the debug supervisor returns to the object program.

The text of the request itself follows immediately after the $DEBUG card. If an invalid or erroneous action is specified in the text of a debug request, that action is deleted. The text consists of procedural statements written in the FORTRAN format.

### Extending the Variable Field of a Card

The variable field of $IBDBL and $DEBUG control cards may be extended over more than one card by following the card with $ETC cards as shown below.

| 1 | 8 | 16-72 |
|---|---|---|
| $ETC | | Extension of variable field |

Columns 16-72 are scanned in the same manner as the card preceding the sETC card.

## Debugging Statements

The statements used in the text of a request are derived from the FORTRAN IV language, with additions and changes made for debugging purposes. Constants, variables, subscripts, arithmetic expressions, and logical expressions that may be used within a debug statement are similar to those used by the FORTRAN IV programmer.

The debugging restrictions and modifications to standard FORTRAN IV usages are listed below.

1. The debugging compiler treats blanks in a statement as terminators. Therefore, no blanks may be imbedded in a character string that is to be treated as a single symbol, constant, operator, or verb. This restriction also applies to debug elements preceded by the character =.

2. At least one blank must follow each statement verb (for example, DUMPbx, where b represents a blank).

3. The operator ** (exponentiation) cannot be used in debug statements.

4. Functions cannot appear in debug statements.

5. The logical constants .TRUE. and .FALSE. cannot be used.

6. A numerical constant may be represented as a real constant, a decimal integer constant, or an octal integer constant. A real constant is indicated by the inclusion of a decimal point, or by the inclusion of the decimal point and the letters E, EE, or D, followed by an exponent. (E indicates single precision, EE and D double precision.) Octal integer constants are differentiated from decimal integer constants as follows: if the first digit of the number is zero and if the number does not contain any non-octal characters (8 or 9), it will be considered octal, and it may be up to 13 digits in length; otherwise it will be considered a decimal constant. Complex constants are of the form $(n_1, n_2)$, where $n_1$ represents the real part of the complex number and $n_2$ the imaginary part. However, unlike the FORTRAN IV representation of a complex constant, $n_1$ and $n_2$ may be octal integers, decimal integers, or real constants.

The following are examples of numerical constants:

| | |
|---|---|
| 0123 | Octal Integer |
| 123 | Decimal Integer |
| 0129 | Decimal Integer |
| 0.129E3 | Real |
| (0120, -129) | Complex |

7. A variable name may be any legal MAP symbol. However, FORTRAN IV conventions will override MAP notation, and those MAP symbols (1.2) that would be treated as numerical constants (as on the right hand side of the equal sign in SET statements) should be redefined prior to use. (See the explanation of the sREDEF card for details on the redefining facility.) The mode of a variable is indicated in a debugging dictionary or NAME statement rather than, as in FORTRAN IV, by its leading character or by the use of a FORTRAN IV Type statement.

8. Subscripts may be any valid arithmetic expressions. Nonintegral subscripts will be truncated to integer values.

9. =An and =Rn, where $n$ is an unsigned octal integer, may be used as elements of expressions. The mode of these elements is octal. (See the "LIST Statement" for a further description of the quantities =An and =Rn.)

10. Elements of expressions may be preceded by deck name qualifiers of the form:

deckname$$element

(See description under "Qualification of LIST Items.")

11. Additional items that may be used as elements of expressions are described in Table 1, in the section "Quantities Available for Use in Debug Request Statements." For full information on expressions, see the following section.

Debug statements are punched in columns 7-72 of the card. Continuation cards are indicated by any character other than a blank or zero in column 6. Comments cards (those with a C punched in column 1) are permitted.

### Arithmetic and Logical Expressions in Debugging Statements

Arithmetic and logical expressions used in debugging statements are modified forms of the FORTRAN IV language.

#### Arithmetic Expressions

An arithmetic expression consists of sequences of constants, subscripted variables, and nonsubscripted variables, separated by arithmetic operation symbols, commas, parentheses, and various special elements.

The arithmetic operation symbols + − * / denote addition, subtraction, multiplication, and division, respectively.

Quantities in arithmetic expressions need not be in the same mode. The hierarchy of modes is as follows:

1. Complex
2. Double precision real
3. Real
4. Octal and decimal integer

Where modes are mixed, the quantities in the expression will be converted to the mode used in the expression that is highest in the hierarchy of modes.

10

Note that if octal integer, decimal integer, symbolic, logical, and/or alphameric modes are specified in arithmetic expressions no conversion will be performed, since these modes are equally low in the hierarchy.

### Logical Expressions

A logical expression consists of sequences of logical variables, which must be separated by logical operation symbols, and arithmetic expressions, which must be separated by relational operation symbols. A logical expression always has the value true or false.

The permissible logical operators (where $b$ represents a blank and $x$ and $y$ are logical expressions) are:

| | | |
|---|---|---|
| bNOTbx | or | b.NOT.bx |
| xbANDby | or | xb.AND.by |
| xbORby | or | xb.OR.by |

The operands can be relational expressions or logical variables. A logical variable is true if its sign is minus, false if its sign is plus.

The permissible relational operators are:

| | | | |
|---|---|---|---|
| bGTb | or | b.GT.b | Greater than |
| bGEb | or | b.GE.b | Greater than or equal to |
| bLTb | or | b.LT.b | Less than |
| bLEb | or | b.LE.b | Less than or equal to |
| bEQb | or | b.EQ.b | Equal to |
| bNEb | or | b.NE.b | Not equal to |
| bLGTb | or | b.LGT.b | Logically greater than |
| bLGEb | or | b.LGE.b | Logically greater than or equal to |
| bLLTb | or | b.LLT.b | Logically less than |
| bLLEb | or | b.LLE.b | Logically less than or equal to |
| bLEQb | or | b.LEQ.b | Logically equal to |
| bLNEb | or | b.LNE.b | Logically not equal to |

The operands can be numerical variables or expressions.

The first six of the above relational operators result in an algebraic comparison between the operands. If they differ in mode type, the conversion is made as described for arithmetic expressions. If the operands are complex (e.g., $(n_1, n_2)$ represents one operand and $(n_3, n_4)$ represents the other), they are considered equal only if $n_1 = n_3$ and $n_2 = n_4$. The values $n_1^2 + n_2^2$ and $n_3^2 + n_4^2$ are used to determine which operand is greater.

The last six of the above relational operators result in a 36 bit unsigned comparison. If the modes are mixed, no conversion is performed.

Parentheses can be used in arithmetic or logical expressions to specify the order in which the operations are to be performed. Where parentheses are omitted, the order of operations is as follows:

* and /
+ and −
relational operations
NOT
AND
OR

## SET (Arithmetic) Statement

The SET statement provides the programmer with arithmetic and logical capabilities within a debug request. The general form of this statement is:

SET s

The letter $s$ is any valid arithmetic or logical statement other than the logical IF statement.

EXAMPLES
SET A=0
SET B=C .GT. 4 (B is true if C is greater than 4; B is false otherwise)

## Logical IF Statement

The debugging logical IF statement is similar to that of FORTRAN IV, with certain additions and restrictions. The general form of this statement is:

IF (t) s
or
IFbtbs

The letter $b$ represents one or more blanks; $t$, any logical expression that contains neither function calls nor the ** operator; and $s$, an unconditional executable statement or an ON statement followed by an unconditional executable statement.

If the logical expression $t$ is true, statement $s$ is executed. If $t$ is false, statement $s$ will not be executed.

EXAMPLES
IF (B EQ A * 3.5) DUMP X, Y, Z
IF A(I, I−J) EQ 3.4E2*QSUM DUMP X
IF (X .EQ. 3 .AND. Z .LT. 24) GO TO 3
IF LOGVAR .AND. (ALPHA+6 LE BETA OR LGVAR1) RETURN

## ON Statement

The ON statement is a count-conditional statement that permits the programmer to control the time when a debugging action is to be taken. It is similar to the FORTRAN IV logical IF statement and is of the general form:

ON [(c)] i, j, k s

The letters $i$, $j$, and $k$ are any arithmetic expressions (if they are not integral, they will be truncated); $c$ is a unique symbol representing a counter name, which should not be contained in the debugging dictionary; and $s$ is an unconditional executable statement or an IF statement followed by an unconditional executable statement.

Statement $s$ is executed the $i$th time through the ON statement and each $k$th time thereafter until $j$ is exceeded. If $j$ is omitted, the statement is executed the $i$th time and every $k$th time thereafter. If $k$ is omitted, it is assumed to be one. If both $j$ and $k$ are omitted, the statement is executed only the $i$th time through the ON statement.

When included, $c$ creates a named counter for the ON statement. The value of $c$ is initially zero; during execution of the ON statement, the counter is incremented by one before being tested. As with other variables, $c$ may be used in any computation or test, and it may be named in other ON statements.

If $c$ is named in other ON statements, $c$ is incremented by one each time any one of the ON statements is executed. The counter can appear on the left-hand side of an expression in a SET statement. The deck name on the SDEBUG card heading the request implicitly qualifies $c$. All references to $c$ that are implicitly or explicitly qualified by this deck name are treated as references to the counter created by $c$. Therefore, if $c$ is duplicated by a symbol in an object deck's debugging dictionary, it will not be possible to refer to that object deck's symbol in the subsequent statement of the debugging packet.

If $c$ is not specified, an unnamed counter is created internally; it cannot be referred to by any other statement.

## GO TO Statement

The statement GO TO n (where $n$ is a decimal integer) is an unconditional transfer to the debugging statement numbered $n$. Statement numbers used in an unconditional GO TO must refer to statements within the debugging packet, *not* to statement numbers in the program being debugged.

## LIST Statement

The LIST statement specifies the storage areas and/or registers that are to be dumped. Its general form is:

statement number LIST element1, element2....

*statement number* is a standard FORTRAN statement number of up to five numeric characters (punched in columns 1-5) and *element* denotes the address of a quantity to be dumped. Any number of elements, separated by commas, may be specified.

The mode in which the data is dumped is determined from the debugging dictionary, or from a NAME statement. If this information is not available, the dumps will be octal unless the mode is specified as in item 2 in the following text.

Except where otherwise indicated "symbol" in the following items is defined either in the debugging dictionary or in a NAME statement.

Elements may be specified in seven ways:

1. Quantity — the specified quantity is dumped. It may be one of the following:

symbol
The array, double-precision number, complex number, or single word denoted by this symbol is dumped.

symbol (subscripts)
The array element, double-precision number, complex number, or single word denoted by this subscripted symbol is dumped. Any symbol may be singly subscripted, but only those symbols that have had dimensions specified may have more than one subscript. The subscripts may be any arithmetic expressions. The mode of the dump is the same as the mode of the symbol. If ALPHA(6) is specified, the contents of the location ALPHA+5 are dumped; however, if ALPHA is double-precision or complex, ALPHA+10 and ALPHA+11 are dumped.

symbol ± n
This causes the single word denoted by this quantity to be dumped. *symbol* is as defined in the preceding text and $n$ is an unsigned decimal integer. The mode of the dump is that of the specified location and is not necessarily the same mode as the symbol.

= Rn
The word at relative location $n$ where $n$ is an unsigned octal integer, is dumped. Only one word is dumped, even though it may be part of a double-precision or a complex number.

= An
The word at absolute location $n$ where $n$ is an unsigned octal integer, is dumped. Only one word is dumped, even though it may be part of a double-precision or a complex number.

The MAP programmer may refer to the sections "Quantities Available for Use in the Debug Request Statements" and "Item Designations for LIST Statements" for additional ways of addressing quantities.

2. (loc1, loc2 [, m]) — this specification causes the region from *loc1* through *loc2* to be dumped in format $m$. *loc1* and *loc2* may be any of the quantities defined above, or a FORTRAN statement number within the source program. The permissible values of $m$ are:

| | |
|---|---|
| O | Octal |
| S | Symbolic instruction |
| F | Floating-point number |
| X | Fixed-point number |
| D | Double-precision number |
| J | Complex number |
| L | Logical |
| H | Alphameric |

If $m$ is specified, it overrides any other mode designations for the quantities involved. If $m$ is not specified, the region is dumped in the mode(s) of each item in the region, as defined in the debugging dictionary.

Thus, if LOC3 is a double-precision array and LOC4 is a floating-point array with dimensions of (3, 10), the specification (LOC3(6), LOC4(3,4), F) causes LOC3+10 through LOC4+11 to be dumped in the floating-point mode. A specification of (6, = R127) causes the region from statement number 6 through relative location 127 (octal) to be dumped in the mode(s) supplied by the debugging dictionary. A specification of (ARRAY1, ARRAY2, X) causes all of ARRAY1, all intervening locations, if any, and all of ARRAY2 to be dumped in the fixed-point decimal mode.

3. (data list) — selected elements of arrays are dumped. This specification operates in a manner similar to the implied DO in FORTRAN IV input/output lists.

*data list* may be one of the following:

symbol (subscripts)    e.g., (A(I), I=1, 4)
(data list, range)    e.g., ((B(I, J), I=1, 4), J=1, 3)
data list, data list...    e.g., (A(I), (C(J, I), J=1, 9, 2),
                        D(I), I=1, 4)

Any arithmetic expression may be used as a subscript. If it is not integral, it will be truncated.

*range* has the following form:

$$v = a_1, a_2 \,[, a_3]$$

The letter $v$ is a subscript in this statement and the $a_i$ are arithmetic expressions. Any other use of $v$ in the program or in the debugging packet will be ignored for this statement. The dumps specified by the data list are taken for $v = a_1$ through $v = a_2$ in increments of $a_3$. If $a_3$ is omitted, it is assumed to be 1. Ranges may be nested to a maximum subscript depth of three.

The following are valid examples:

(A(I,I), I=1, 3)
dumps A(1,1), A(2,2), A(3,3)
(B(I, 6−I), I=1, 5, 2)
dumps B(1, 5), B(3,3), B(5, 1)
((A(I,J), I=1, 3), J=6, 8, 2)
dumps A(1, 6), A(2, 6), A(3, 6), A(1, 8), A(2, 8), A(3, 8)
((C(2*J−4, 10−3*), I=1, 3) J=3, 4)
dumps C(2, 7), C(2, 4), C(2, 1), C(4, 7), C(4, 4), C(4, 1)
(A(J, J), (C(2*J−4, 10−3*I), I=1, 3), J=3, 4)
dumps A(3, 3), C(2, 7), C(2, 4), C(2, 1), A(4, 4), C(4,7),
      C(4, 4), C(4, 1)

Because the following example contains four nested ranges, it is invalid:

((((A(I, J, K), B(I, J, L), I=1, 2), J=1, 2), K=1, 2),
L=1, 2)

The statement LIST (A(I, J), I=1, 3) specifies A(1, J), A(2, J), and A(3, J), where J is a variable defined previously either in the source program or in the debugging packet.

4. // or /control-section name/ — blank COMMON or the specified control section is dumped. If the /control-section name/ option is specified, the debug preprocessor uses the following as the length of the control section: the length of the first real section of the name encountered, even if $NAME, $USE, or $OMIT cards cause this section to be deleted in favor of another of different length at object time.

5. //±n or /control-section name/±n — the word that is dumped is at the beginning of blank COMMON ±n or at the beginning of the specified control section ±n, where $n$ is an integer. FORTRAN labeled COMMON names are considered control-section names.

6. PROGRAM — the entire object program and all system subroutines used by the object program, as well as file text, buffers, and blank COMMON, are dumped.

7. 'message' — a message is printed as specified. The message itself should not exceed the number of characters allowable for a single printed line. It cannot contain quotation marks. The external quotation marks, however, are required. Two quotation marks will yield a blank line.

8. CONSOLE — the contents of the AC, MQ, index registers, entry keys, sense switches, divide check, and overflow indicators are dumped.

Additional list items are described in Table 1, in the section "Quantities Available for Use in Debug Request Statements."

### Qualifications of LIST Items

Unless explicitly qualified, all symbols in a debug request are implicitly qualified by the deck name of the $DEBUG card heading the request.

To allow for communication between decks, any item or set of items may be qualified with the deck name and two dollar signs, as shown:

    deckname$$item
    deckname$$(item, item...)

The *items* specified are as defined for LIST elements. For example, A$$B refers to symbol B in deck A, and AB$$(BB, CB, (RB, SB, O) )refers to items BB, CB, and the octal region RB through SB, which are all in deck AB.

## DUMP Statement

The DUMP statement dumps the quantities to be printed as debugging output. It is similar in structure to the FORTRAN IV WRITE statement and is of the general form:

    DUMP list

*list* is a series of items that is either a direct reference to the data to be dumped or the statement number(s) of LIST statements specifying the data to be dumped. The acceptable data specifications for both direct references and LIST statements are itemized in the discussion of the LIST statement.

The DUMP statement causes information to be written on the debug work unit (DWU). The postprocessor edits the data on the DWU and writes it and the dump number on the system output unit. The formats supplied for the list items of the DUMP statements are:

1. The symbolic reference of the item, along with its location(s), deck name, and control section, is written to identify the debugging output. (The relative location may not agree with that of the MAP assembly because of the deletion of control sections and adjustment caused by the use of the EVEN pseudo-operation.)

2. The value(s) of the item is written. The format, which is based on the number of elements in the item

and the mode of the item (as listed under point 2 in the section "LIST Statement"), is derived as follows:

| TYPE | NO./LINE | FORMAT |
|---|---|---|
| Octal | 4 | OP xxxxxx xxxxxx |
| Symbolic Instruction | 2 | ±x xxxxx x xxxxx OP A, T, D[1]. |
| Floating-Point | 6 | ±.xxxxxxxx±xx |
| Fixed-Point | 6 | ± xxxxxxxxxxxx. (leading zeros dropped) |
| Double-Precision | 4 | ±.xxxxxxxxxxxxxxxxD±xx |
| Complex | 3 | ±.xxxxxxxx±xx ±.xxxxxxxx±xxJ |
| Logical | 8 | .TRUE. or .FALSE. |
| Alphameric | 96 char | xxxx...xxx |

## CALL Statement

The CALL statement is used to call subroutines. Its general forms are:

        CALL    subr
        CALL    subr (arg1, arg2,...)

where *subr* is the name of the subroutine being called and *arg1, arg2, . . .* are the arguments. The subroutine that has been called must use neither the CHAIN facilities of IBJOB nor call S.SLDR; otherwise, overlay might be induced. *subr* must be a real section name. The subroutine must be in one of the input decks if the job is a chain job. Otherwise it may be loaded from IBLIB. Output arguments to a subroutine that has been called may not be constants, any of the quantities listed in Table 1, or quantities that involve address computation.

## RETURN Statement

The RETURN statement causes a return to the interrupted program. There is an implicit RETURN at the end of each debug request.

## PAUSE Statement

This statement causes the message "DEBUG REQUEST PAUSE" to be printed, and processing to be suspended. When START is pressed, execution continues with the next statement in the request.

## STOP and CALL EXIT Statements

The STOP and CALL EXIT statements terminate execution by transferring to S.JXIT.

## NAME Statement

The NAME statement permits the programmer to define symbols for use within debug requests. It also defines symbols where no debugging dictionary, or an insufficient one, has been supplied.

---

[1]OP A, T, D refers to the symbolic representation of a machine language instruction. This is primarily of interest to the MAP programmer.

The general form of the NAME statement is:

$$\text{NAME } symbol_1/ \begin{Bmatrix} location \\ =NEW \end{Bmatrix} \Big[ (mode\ [(dimensions)]) \Big]$$
$$[, symbol_2/..]$$

The parameters are:

**symbol₁**
Any valid MAP symbol.

**location**
A location designation. It may be a nonsubscripted symbol (plus or minus an integer, if desired); =Rn (relative location), where *n* is an unsigned octal integer; or =An (absolute location) where *n* is an unsigned octal integer.

**=NEW**
A location of octal format or an area corresponding to the specified mode and dimensions is to be generated for the symbol.

**(mode)**
The mode of *symbol*. It may be O, F, X, D, J, L, S, or H, as described in the discussion of the LIST statement.

**(dimensions)**
The dimensions, if any, of the array denoted by *symbol* are specified as:

$$(d_1, d_2, d_3)$$

Arrays are structured and referred to exactly as they are in FORTRAN IV.

The FORTRAN programmer will primarily use the =NEW option of the NAME statement.

The deck name on the $DEBUG card heading the debug request in which a NAME statement is found qualifies implicitly the symbol defined in the NAME statement. The NAME statement defines the symbol only for subsequent uses of that symbol (which has been implicitly or explicitly qualified by the deck name on the $DEBUG card).

This also applies to =NEW symbols. The same symbol may not appear more than once under the =NEW option; that is, the user should not use the =NEW option in more than one NAME statement for the same symbol. By the same token, he may not define the same symbol in NAME statements more than once for a single object deck.

## Redefining Symbols

The $REDEF card allows the user to change the names of symbols used in source decks to make them acceptable for use in debug requests. Unacceptable symbols are those MAP symbols that would be interpreted as numbers by the debug compiler (e.g., 0.1E3, 4.6EE2, 633.D4, 1.2). The general form of the $REDEF card is:

| 1 | 8 | 16-72 |
|---|---|---|
| $REDEF | deckname | old₁bASbnew₁ |
| | | [, old₂bASbnew₂ . . .]. |

*deckname* is the name of the deck in which the symbol to be redefined appears; *b* represents one or more

blanks; the $old_1$ are the symbols to be redefined; and the $new_1$ are the new names of the symbols.

The variable field (columns 16-72) of a $REDEF card may be extended to more than one card by using the $ETC card as described for the $DEBUG card in the section "Load-Time Debug Requests."

$REDEF must be followed immediately by a $DEBUG card or another $REDEF card.

## Debugging Dictionary

The debugging dictionary provides communication between the program being debugged and the debugging package. This dictionary contains symbols, with their relative locations, modes, dimensions, and mode changes. It also contains FORTRAN statement numbers and their relative locations.

The debugging dictionary is requested by the user on a $IBMAP or $IBFTC control card and, when requested, is produced by IBMAP. The general form of the control card is:

```
1              8          16-72
$IBMAP
$IBFTC         deckname    other options
                           [, debug dictionary options]
```

The card name, deck name, and other options are as given in the publication *IBM 7040/7044 Operating System (16/32K): Programmer's Guide*, Form C28-6318.

The debugging dictionary options are:

NODD

No debugging dictionary. This is the standard option, which specifies that a debugging dictionary is not desired. If no option is specified, NODD will be assumed.

DD

Full debugging dictionary. All mode changes and symbols used in the MAP program will appear in the debugging dictionary. In a FORTRAN IV program, all statement numbers, all variable names, and all symbols generated by IBFTC will be included.

SDD

Short debugging dictionary. Only those symbols specified by the MAP pseudo-operation KEEP (described in the section "KEEP Pseudo-Operation") will appear in the debugging dictionary. If this option is specified on a $IBFTC card, IBFTC will generate KEEP operations for statement numbers and program variable names.

### Supplying Modal Information to the Debugging Dictionary

IBFTC supplies modal and dimensional information to IBMAP and then, automatically, to the debugging dictionary. When double-precision (D) or complex (J) modes are specified, the dimensions indicate the number of two-word entries contained in the array. (The product of these numbers will equal half the value field of the BSS.) However, the MAP programmer must supply this information himself when it cannot be assumed by the nature of the instruction, i.e., when he is using BSS, BES, EQU, and SYN. (Only modal information may be given with a BES; dimensional data will be ignored.) This information is specified in additional subfields of the variable field of these operations, as:

| NAME FIELD | OPERATION FIELD | VARIABLE FIELD |
|---|---|---|
| Symbol | $\begin{Bmatrix} \text{BSS} \\ \text{BES} \\ \text{EQU} \\ \text{SYN} \end{Bmatrix}$ | Value $\left[ \text{, mode} \quad [(d_1, d_2, d_3)] \right]$ |

*symbol* and *value* are the standard forms for these pseudo-operations; *mode* is O, F, X, D, J, L, S, or H, as described in the discussion of the LIST statement; and $d_1$, $d_2$, and $d_3$ are the dimensions, if any, of the array denoted by the symbol. (Arrays are structured and are referred to as in FORTRAN.)

EXAMPLES

| A | BSS | 25, F(5, 5) |
|---|---|---|
| B | EQU | A + 10, F(5, 3, 2) |
| C | SYN | A + 6, X |
| D | BES | 2, X |

### KEEP Pseudo-Operation

The KEEP pseudo-operation permits the MAP programmer to specify, in his source program, a debugging dictionary that contains only those symbols he wishes to use in debug requests. The format of the KEEP pseudo-operation is:

| NAME FIELD | OPERATION FIELD | VARIABLE FIELD |
|---|---|---|
| Blanks | KEEP | One or more symbols separated by commas. |

The symbols in the variable field are entered into the debugging dictionary along with any modal and dimensional information that was supplied in BSS, BES, EQU, and SYN pseudo-operations. Any number of KEEP pseudo-operations may appear in a program. If the NODD or DD option was specified on the $IBMAP card, the KEEP pseudo-operation is ignored.

Note that qualified symbols are not entered into the debugging dictionary. The first encountered of the same named symbols is the one entered into the debugging dictionary.

## Additional Load-Time Debugging Features

The following sections contain descriptions of additional debugging facilities that are of interest mainly to the MAP programmer, but may also be used by the FORTRAN or COBOL programmer.

### Quantities Available for Use in Debug Request Statements

The quantities in Table 1 are available for use in statements that make up the text of a debug request. There

should not be a blank between the equal sign and the quantity.

Table 1. Special Quantities Available for Use in Statements

| QUANTITY | DEFINITION |
|---|---|
| = AC | Accumulator $(S, 1, 2, \ldots, 35)$. |
| = AC $(i_1 - i_2)$ | Accumulator bits $i_1$, through $i_2$; $0 \leq i_1 \leq i_2 \leq 35$; bit $0 = $ S-bit. |
| = LAC | Logical accumulator $(P, 1, 2, \ldots, 35)$. |
| = LAC $(i_1 - i_2)$ | Logical accumulator bits $i_1$ through $i_2$; $0 \leq i_1 \leq i_2 \leq 35$; bit $0 = $ P-bit. |
| = MQ | Multiplier-quotient register $(S, 1, 2, \ldots, 35)$. |
| = MQ $(i_1 - i_2)$ | Multiplier-quotient bits $i_1$ through $i_2$; $0 \leq i_1 \leq i_2 \leq 35$; bit $0 = $ S-bit. |
| = XRn | Index register n; $1 \leq n \leq 7$. |

NOTES:

1. The above quantities in arithmetic or logical expressions have the value of the registers upon entry to the debugging routines or the values subsequently assigned by their appearance on the left-hand side of a SET statement. The values of =XR3, =XR5, =XR6, and =XR7 are the logical OR's of index register 1, 2, and/or 4. Reference to these values does not automatically reset the index registers.

If the above quantities (except =XR3, =XR5, =XR6, or =XR7) appear on the left-hand side of a SET statement, the register contains the newly assigned value when subsequently dumped or referred to in a DEBUG statement. The register will also contain this newly assigned value upon re-entry to the object program unless the register specified was =LAC, =LAC($i_1-i_2$), XR3, XR5, XR6, or XR7. Note that =LAC and =AC refer to separate locations in the /DEBUG deck; setting one does not affect the other.

The mode of an index register quantity in a statement is integer (i.e., fixed-point). The other register quantities are of indeterminate mode; they are never converted and the values of other variables in the statement are not converted to agree with them.

2. When the user specifies a bit extraction of AC, LAC, or MQ, the bits extracted will be right-justified and will be dumped under the heading /DEBUG+n (where n is a relative location).

The following examples of the logical IF clause illustrate the use of some of the quantities that refer to internal registers.

IF ( =LAC OR =MQ )
IF ( =XR4−8 EQ 3*=XR2 )

**Address Computation**

Considered as a whole, an expression represents an effective address just as does any individual element. The effective address is determined through the chaining effect of a series of sequential steps, each step being the effective address upon which the succeeding step is to operate.

The form for an address expression is:

$$=\text{C (effective address)} \begin{bmatrix} \left\{ \begin{matrix} \pm \\ * \\ / \end{matrix} \right\} \end{bmatrix} =\text{C (effective address)} \begin{bmatrix} \left\{ \begin{matrix} \pm \\ * \\ / \end{matrix} \right\} \ldots \end{bmatrix} )$$

An effective address is of the form:

$$\text{base address} \begin{bmatrix} \left\{ \begin{matrix} \text{ADDR} \\ \text{DECR} \\ (i_1 - i_2) \end{matrix} \right\} \end{bmatrix} \begin{bmatrix} , \text{COMPL} \end{bmatrix}$$

The *base address* may be any of the following:

1. Any subscripted or nonsubscripted variable defined in the source deck, to which reference is currently being made

2. Any of the special names ( =XRn, =AC, etc.) specified previously. These special names must be followed by one of the extraction options described below [i.e., ADDR, DECR, or $(i_1-i_2)$]

3. A relative ( =Rn) or an absolute ( =An) address

4. A decimal integer

5. A statement number (FORTRAN only)

6. Another effective address expression using any of the four items above as a base address and the operators described below

Components of the operator string may be any of the following terms:

| OPERATOR | DEFINITION |
|---|---|
| ADDR | Use, as the next effective address, the address portion of the word specified by the current effective address. |
| DECR | Same as above, using the decrement portion. |
| $(i_1 - i_2)$ | Same as above, using bits $i_1$ through $i_2$ for $0 \leq i_1 \leq i_2 \leq 35$. |
| COMPL | Complement the current effective address to form the next effective address. |
| + argument | Add the specified argument, which can be any of the forms defined for a base address, to the current effective address to form the next effective address. |
| − argument | Same as above, but subtract. |
| * argument | Same as above, but multiply. |
| / argument | Same as above, but divide. |

The following are examples of address computation:

| | |
|---|---|
| =C(A) | The address of A |
| DUMP =C(A) | Same as DUMP A |
| SET =C(A)= =C(B)+2 | Same as SET A=B+2 |
| =C(A,ADDR) | The address portion of the word at location A |
| =C(A(5), DECR, COMPL, +A(3)) | The complement of the decrement portion of A(5), plus A(3) |
| =C(A(7), DECR, + =C(A(7), ADDR), −1) | The decrement portion of A(7) plus the address portion of A(7), minus one |

**Bit Extraction**

In arithmetic expressions (e.g., in SET, IF, CALL, or ON statements and also in address computation and in subscripts) it is convenient to be able to handle partial word operations. The notation to be used is:

=C (address computation) (bit specification)
or
=mr (bit specification)

where the bit specification is $i_1$-$i_2$, ADDR, or DECR, and *mr* (machine register) is AC, LAC, MQ, or XRn;

$$=mr(i)$$

where $0 \leq i \leq 35$ is also permitted.

The following are examples of partial word operations:

| | |
|---|---|
| =C(A) (18-20) | The tag of A |
| =AC(0) | The sign of the AC |
| =MQ(17) | Bit 17 of the MQ |

The following statement replaces the decrement of the word at A with the sum of 6 and the tag of B:

$$SET = C(A) (DECK) = = C(B) (18-20) + 6$$

The following statement replaces the address portion of the AC with the address portion of the word at Q (the rest of the AC is not affected) if the MQ bits 4 and 6 are 1, and bits 5 and 7 are 0:

$$IF = MQ (4-7) EQ 012 SET = AC (ADDR) = Q$$

The value of an item in an expression or output list on which bit extraction has been performed consists of those bits right-adjusted with zeros in the high-order positions. Note that the logical value of such an item is always false if the number of bits extracted is 35 or fewer. (Any type conversion necessary is done after the extraction.)

## Location Values

When manipulating instructions, the MAP programmer often needs the value of a symbol rather than the contents of the location addressed by the symbol. The programmer uses the notation =V (symbol) to designate the value of a symbol.

EXAMPLE

$$SET = C (Q, ADDR) = = V(S) + 1$$

This replaces the address field of Q with one plus the location value of S.

## Example of a Load-Time Debugging Packet Used with a FORTRAN IV Program

Figure 2 contains an example of a load-time debugging packet for a FORTRAN IV program. The numbers in the first line across indicate the card columns in which the various fields begin.

The $IBDBL card heading the packet calls in the debugging compiler. It specifies that all debugging activity is to cease when 450 debug requests have been executed at object time, that a maximum of 500 lines of debugging output is to be printed, that the FORTRAN Input/Output Subroutines are to be used to write
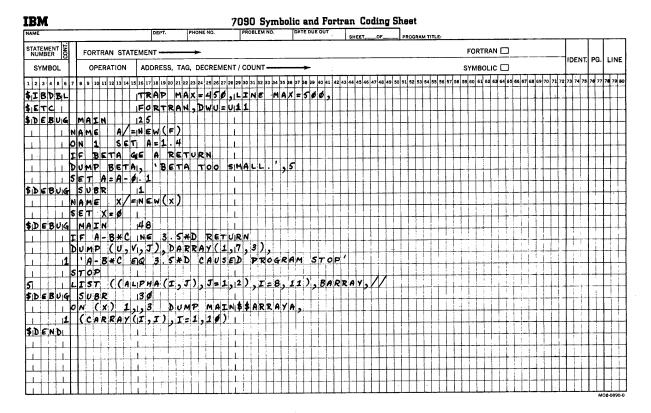


Figure 2. Symbolic Debugging — FORTRAN Example

```
NAME                      DEPT.      PHONE NO.      PROBLEM NO.    DATE DUE OUT
                                                                          SHEET___OF___   PROGRAM TITLE:

STATEMENT                 FORTRAN STATEMENT ──────►                                    FORTRAN □
 NUMBER                                                                                                        IDENT. PG. LINE
  SYMBOL        OPERATION   ADDRESS, TAG, DECREMENT / COUNT ──────►                    SYMBOLIC □

$IBDBL               TRAP  MAX=1000,LINE MAX=500,
$ETC
$DEBUG   DECKA    B4
         ON  1,10  DUMP 2,3
$DEBUG   DECKB    START+7
         IF (LOGVAR) ON 3,18,3  DUMP X,/CNTRA/,
      1  (=R6,=R103,O)
3     LIST QUAN,//
$DEBUG   DECKA    B4
         ON 100 GO TO 10
         RETURN
10       DUMP 2,3
         STOP
2     LIST CONSOLE
$DEBUG   DECKB    RESTRT
         ON 1,10,2  IF QUAN EQ 0
      1  DUMP 2,
      2  'QUAN EQUAL TO ZERO',6
6     LIST (RESTRT,RESTRT+99,S)
$DEBUG   DECKA    B7,A4
         IF (A .LT. B) RETURN
         DUMP A,B
$DEND
```

Figure 3. Symbolic Debugging — MAP Example

dump markers (on FORTRAN logical unit 6) and that utility unit 11 is to be used for the debugging work unit.

The first debug request is executed immediately before statement 25 in deck MAIN. The real variable A is defined for use in the debugging requests for that deck. The first time statement 25 is executed, A is set to 1.4. BETA (in program MAIN) is tested before each execution of statement 25. If BETA is less than A, BETA and the elements described by LIST statement number 5, contained in the third request, are dumped; the message BETA TOO SMALL is printed; and A is decreased by 0.1. If BETA is greater than or equal to A, return is made to the interrupted program; then statement 25 is executed.

The second and fourth debug requests are executed at statements 1 and 30, respectively, in deck SUBR. Suppose that deck MAIN calls SUBR several times and that statement 1 is the first executable statement in SUBR. Further, suppose that SUBR contains a DO loop that causes statement 30 to be executed several times. Under these conditions, the second request sets the counter X to 0 each time that SUBR is called; the fourth request dumps array ARRAYA in program MAIN, the first, fourth, seventh, etc., times that statement 30 in SUBR is executed; in addition, it dumps the principal diagonal of the matrix CARRAY (in deck SUBR). Thus,

the second and fourth requests trace the initial action of SUBR each time it is called.

The third request is executed immediately before each execution of statement number 48 in MAIN. The value of A−B*C is tested against 3.5*D, and if the two values are not equal, control returns to the program. If they are equal, the following are dumped:

1. The area (in complex number format) from U through V (if V is an array, the entire array will be dumped).

2. DARRAY (1, 7, 3) and the message A−B*C EQ 3.5*D CAUSED PROGRAM STOP. Program execution then stops and the debug postprocessor is called.

The third request also contains LIST statement number 5, which, when used in a DUMP statement, causes the following information to be dumped: ALPHA (8, 1), ALPHA (8, 2), ALPHA (9, 1), ALPHA (9, 2), ..., ALPHA (11, 2); all of the elements of BARRAY; and all of blank COMMON.

The $DEND card terminates the action of the debugging compiler.

## Example of a Load-Time Debugging Packet Used with a MAP Program

Figure 3 illustrates a load-time debugging packet for a MAP program. The numbers in the first line across indicate the card columns in which the various fields begin.

The sIBDBL card heading the packet specifies that all debugging activity is to cease when 1,000 requests have been executed and that a maximum of 500 lines of debugging output is to be printed. It also specifies that the object program is using the Output Editor (JOBOUL) and that the Debugging Work Unit is assumed to be the system checkpoint unit.

The first request specifies that the elements in LIST statement numbers 2 and 3 (which appear later in the packet) are to be dumped the first time and each time thereafter up to but not including the eleventh time that the instruction at B4 in DECKA is to be executed.

The second request is executed immediately preceding each execution of the instruction at START+7 in DECKB. On the third, sixth, ninth, twelfth, fifteenth, and eighteenth times that logical variable LOGVAR is true (i.e., sign is nonzero), the following will be dumped: X; control section CNTRA; and the region from relative locations 6 (octal) through 103 (octal) in octal format. The second request also contains LIST statement 3, which, when used in a DUMP statement, causes the quantity QUAN and the blank COMMON (//) control section to be dumped.

On the hundredth time that location B4 in DECKA is reached, information specified by the third request is dumped, program execution is terminated, and the debug postprocessor is called. At all other times, control is returned to the interrupted program. The information dumped consists of the status of the console registers (LIST statement number 2), and the elements specified in LIST statement number 3, contained in the second request.

The fourth request causes a test to be made the first time and every second time thereafter, until the ninth time that the instruction at RESTART in DECKB is to be executed. It tests QUAN, and if QUAN is zero, the following are dumped: the elements in LIST statement number 2 (contained in the third request), the message QUAN EQUAL TO ZERO, and the elements in LIST statement number 6 (which specifies the region from RESTRT through RESTRT +99 in symbolic format).

The fifth request is executed immediately before each execution of statements B7 and A4 in DECKA. If A is less than B, control is returned to the interrupted program; otherwise, A and B are dumped.

# Dump Program

The IBM 7040/7044 Operating System contains a general diagnostic and Dump program that provides error messages, error dumps, and debugging dumps during operation of the system. Assembled with this program is a series of dump parameters, identified by five-digit error numbers, that permits the system program to specify the error message and extent of the dump to be provided when a specific error condition occurs.

The Dump program contains terminal diagnostic facilities for system programs, for machine conditions resulting in a trap under error conditions, and for Input/Output Control System error conditions for which no error return has been specified. When such a system error occurs, the console panels and a portion of core storage is saved, and the Dump program is loaded automatically to give a partial or full core storage dump and/or a selective dump of external storage.

To preserve core storage space, all terminal error messages are carried in the Dump program rather than in the Nucleus or the Input/Output Control System.

The Dump program appears in the System Library as four phases or core loads. When loaded, it overlays the higher levels of IOCS. After the dump is finished, the combined monitors and a new IOCS are brought in and control is released to the Supervisor, unless the return is specified. If the return is specified, core storage is restored.

## Calling Sequence

An object program may relinquish control to the Dump routine in the Nucleus by means of the following calling sequence:

```
SXA      *+3,4        (or the equivalent)
TSX      S.SDMP,4
pfx      ret,t,errno
PZE      **
```

The prefix *pfx* is interpreted as follows:

Sign Bit = 1    Pause before returning
Bit 1    = 1.   Dump system panel
Bit 2    = 1    Traceback

S.SDMP is the entry point to the Dump routine in the Nucleus. The location *ret,t* is the point to which control returns upon completion of the dump. The *errno* is a five-digit decimal number that is used to identify the series of parameters describing the error message desired, the special editing routines, and the portions of internal and external storage to be dumped. Upon

completion of the dump, control is returned to the Supervisor, and processing resumes with the next $JOB card unless *ret,t* is non-zero.

A means of obtaining a terminal core dump is to execute the following instructions:

```
SXA      *+3,4
TSX      S.SDMP,4
PZE      ,,13000
PZE      **
```

*errno* 13000 may be used to obtain a full core storage dump in octal with mnemonics and BCD.

## Format

Depending upon the parameters assembled with the Dump program, a call to the Dump program results in one or more of the following types of output:
1. On-line message on the console typewriter
2. Off-line message on the system output unit
3. A dump on the system output unit

The dumps of the console panel and the system panel, the core storage dump, and the external storage dump on the system output unit may be obtained individually or in any combination. The storage dumps may be given in any combination of the following five options:
1. Octal dump (single line)
2. BCD dump (single line)
3. Octal dump with BCD (double line)
4. Octal dump with mnemonics (double line)
5. Octal dump with mnemonics and BCD (double line)

### On-Line Message

The on-line message consists of a five-digit message number that is included in the calling sequence and of a briefly worded message. The phrase JOB TERMINATED is appended by the Dump program if no return is provided.

### Off-Line Message

The off-line message consists of the five-digit message number, the BCD contents of location S.SCUR, the BCD contents of location S.SFAZ, the location of the TSX instruction used in the calling sequence, and the complete error message provided in the dump parameters. This appears in the heading above the console dump.

### Dump Output

An example of an octal dump with mnemonics and BCD characters is shown in Figure 4.

20

```
    ERROR LOCATION  00103
       OPERATOR CALL TO DUMP

       INDICATORS                SENSE SWITCHES              XR1       XR2       XR4

       CFL   DCT          1    2    3    4    5    6       00001     00014     57625

       ON    OFF         OFF  OFF  OFF  OFF  OFF  OFF     -77777    -77764    -20153

             AC                        MQ

          0000001000001            C00000000000C

ATTACHED DEVICES

                    SYSTEM UNITS TABLE              SYSTEM CONTROL BLOCK              UNIT CONTROL BLOCK

S.SLB1              327   C 01135 C 00571      1135   04 04 73 0 00327       571  0 0000 0 0 02201
                                                      412000 0 05134              000000 0 00037
                                                      000001000000              4 00000 0 00327
                                                    0 00000 0 00000            0 00000 0 00000
                                                                              0 00000 0 00000
                                                                              0 00000 0 00606
                                                                              0 00000 0 00000
                                                                              0 00000 0 00000
                                                                                000000000000

S.SIN1              331   C 01035 0 00525      1035   04 03 74 0 00331       525  0 0000 3 0 01210
                                                      410000 0 07006              000000 0 00012
                                                      000001000000              4 00000 0 00331
                                                    0 00000 0 00000            0 00000 0 00000
                                                                              0 00000 0 00000
                                                                              0 00000 0 00012
                                                                              0 00000 0 00000
                                                                              0 00000 0 00000
                                                                                000000000000

DETACHED DEVICES
                 S.SLB2
                 S.SIN2
                 S.SOU2
                 S.SPP2
                 S.SU07
                 S.SU31
                 S.SU32
                 S.SU33
                 S.SU34
                 S.SU35

00000   042000000000  -012141410011   0020CC004121   000000C00000   002060000003   000000000000   000000000000   002000004161
        HPR   4+0000        JAJJ09     TRA   0+00JA         000000   TRA * 0+ 003         000000         000000   TRA   0+00JJ

00010   002000000011   002C60000000   C000C1003560  -162700003026   000001003560  -162700003026   000001004625  -16270C003026
        TRA   0+0009   TRA * 0+ 00C         0010*   TSL   *GOOHF         0010*   TSL   *GOOHF         00100E   TSL   *GOOHF

00020   000000000000  -16270C003C26   0000CC00CC00  -162700003026   374521200000   000701050603   000000000000   000000000000
              000000   TSL   *GOOHF         000000   TSL   *GOOHF   TXH   *NA+00         071563         000000         000000

00030   000000000000   0C0C00000000   000010002725  -176000000014   002000004154   000000000000   000000000000   000000000000
              000000         000000         0080GE   ICT * 000'     TRA   0+00J*         000000         000000         00CC00

00040   002000004210   000C0C0C0C00   0020CC004223   000000000000   000000000000   000000000000   000000000000   000000000000
        TRA   0+00K8         000000   TRA   0+00KC         000000         000000         000000         000000         00000

00050   076400002221   200001100054   0420CC00C052   077400100012   076200002221  -054000000101   006100000056  -002200000050
        BSR   7U008A   TIX   +0180*   HPR   4+000-   AXT   7(08C0   RDS   7S008A   RCH8  N-0011   TCOB  0/000*   TRCB  -B000J

00060  -053400100101   063400100064   0535@0100101   100001100064   050000100102   062100000101  -012060000101   062200001D1
        LXC   N)0811   SXA   6)080U   LAC   5*C811   TXI   80180U   CLA   500812   STA   6A0011   TMI * J+ 011   STD   6B0011

00070   002000000053   000C00000000   000000000000   000000000000   000000000000   000000000000   000000000000   000000000000
        TRA   0+000$         000000         C00000         000000         000000         000000         000000         00000

00100   000020000000   052700002232   0634C0400105   007400400137   223420000000   000000057625   050000000135  -170400000021
              00+000   XEC   5800B+   SXA   6)0-15   TSX   0(0-1*   TIX   B)+000         0005*E   CLA   50001*   TMT   *400DA

00110   076000000004  -076300000044   060200000054   040000000050   060200000050  -032000000117   076700000011  -073400107000
        ENK   7 0004   LGL   PTCOOM   SLW   62C00*   ADD   40000Q   SLW   62000Q   ANA   L+001*   ALS   7X0009   PDX   PJ08YO

00120  -050000100135   077400100003  -162301100060   077100000006  -162300100060   077100000006   200001100122   002000000053
        CAL   N0081*   AXT   7(0803   SAC1  *C180   ARS   7Z0006   SACO  *C080   ARS   7Z0006   TIX   +0181B   TRA   0+0006

00130   002060640542  -002400634541   0C24C0620541  -002200614540   312245642360   002000001407   002000002073   002000002153
        TRCE  0FOU5K   TRCC -DOTNJ   TRCC  0DOS5J   TRCB -BO/N-   TXH   IBNUC   TRA   0+00'7   TRA   0+00+,   TRA   0+00A&

00140   002000002235   002C00400002   C020C0002251   002000002065   076000000161   002000400001   002000002723   002000002571
        TRA   0+00B*   TRA   0+0-02   TRA   0+C0BR   TRA   0+00+V   SWT   7 001/   TRA   0+0-01   TRA   0+00GC   TRA   0+C0EZ

00150   002000002651   002C0C0C2701   C020CC002700   002000002663   002000002662   002000400003   002000004251   002000007157
        TRA   0+00FR   TRA   0+00G1   TRA   0+00G0   TRA   0+00FT   TRA   0+00FS   TRA   0+0-03   TRA   0+00KR   TRA   0+00Z*

00160   002000010364   0C2000010372  -162700010607  -162700010607  -162700010607  -162700010607  -162700010607   002000010607
        TRA   0+013U   TRA   0+013=   TSL   *GC167   TSL   *G0167   TSL   *G0167   TSL   *G0167   TSL   *G0167   TRA   0+013V

00170   002000010373   002C0C011731   0020C0011656   002000011650   002000011667   002000012001   000020000000   010000077777
        TRA   0+013,   TRA   0+01*I   TRA   0+01**   TRA   0+01*Q   TRA   0+01*X   TRA   0+01+1         00+000   TZE   1007#*

00200   012224010364   000001010000   0C07C1734660   000000000001   000005002414   000005002421   000044000340   00017001675
              1BD13U         001100         071,0         C00001         0050D'         0050DA         00M03-         01YO**

00210   000016002126   000701100603   0C06C3011111   000000000000   000000000000   000261077515   100000000006   312262706260
              00*0AF         071863         063199         000000         000000         02/7**   TXI   800006   TXH   IBSYS
```

Figure 4. Octal Dump Format

The page heading is always provided if any dump parameters exist for the message number. Fields in the page heading pertaining to the time of day or elapsed time are printed only when the interval timer is in operation.

The console panel is always provided if any dump parameters exist for the message number. This information includes:

1. The octal contents of location s.SCLK

2. The BCD contents of location s.SDAT

3. The number of minutes for which the current job has run

4. The contents of the registers

5. The settings of the sense switches and the entry keys

6. The settings of the Overflow and Divide Check indicators

The system panel is an option that provides the unit control blocks and system control blocks for each attached unit in the Symbolic Units Table.

The format of the core storage dump is specified by the dump parameters.

The external storage dump is another option that can be specified in the dump parameters. The following information is provided:

1. Symbolic unit containing information to be dumped

2. Type of device, channel, and unit, all identical to information provided in the ATTACH macro-instruction during system assembly

3. Machine interface, select address, and subaddress

4. Number of the file containing information to be dumped on the system unit

5. Number of the record within the file to be dumped

6. Number of words in the record (in octal)

7. Number of words in the record (in decimal)

The mode of each record dumped is indicated by one of the following codes:

| RECORD IN BCD | Binary coded decimal |
| RECORD IN BIN | Binary |
| RECORD IN RDN | Redundancy error (unreadable) |

If there is an indication that the record is in the wrong mode, this record and subsequent records are read in the alternate mode. If a record is redundant, it is listed in the mode in which it was requested, the error is indicated, and the mode is not switched. However, certain devices give no indication of whether the record was BCD or binary.

Errors that occurred on the units that were dumped are noted in the listing. Errors on units used by the Dump program are noted on the typewriter and in the listing, and are ignored.

## Message Code Numbers

The message code numbers that identify each set of parameters consist of five digits in the following format:

*Ten Thousands Position:*

1. Message without pause.

2. Pause that requires a specified manual action before continuing

3. Pause that requires a choice of specified manual actions before continuing

The 0 code is not used as there are no dead-end halts in the Operating System.

*Thousands Position:*

0. For IBM use only

1. The general policy is that a 1 will not be used in this position until all possible uses of 0 in combination with the hundreds position have been exhausted

2. For IBM use only

*Hundreds Position:* (This is used to designate the programming system in which the error occurred.)

1. IOEX, IOOP

2. IOLS, IOBS

3. Sort/Merge

4. Utilities

5. Supervisors

6. Simulators and Peripheral Programs

7. FORTRAN IV compiler

8. COBOL compiler

9. Macro Assembly Program or Loader

*Tens and Units Positions:* (These positions are to be used to ensure that every code format will be unique and are left to the discretion of the programmer.)

| X1500 — X1549 | System Editor |
| X1550 — X1599 | Processor and Input/Output Editors |
| X1900 — X1999 | Loader (IBLDR) |

*General Purpose Assignments:*

| 10000 | The message OPERATOR CALL TO DUMP |
| 13000 | No Message |

## Dump Parameters

Associated with each error number is a sequence of parameters assembled in IBDMP2. The parameters include one word that identifies the error, a sequence of words that describe the error message to be typed, a series of words that describe the error message to be listed on the system output unit, and words that describe the portions of core storage or external storage media to be dumped.

### Identifier

The word that identifies the error has the following format:

                    PZE          pointr,,errno

where:

pointr
    is the location of the identifier of the next error sequence and is equal to zero to terminate a sequence of pointers.

errno
    is the five-digit error number.

### Error Messages

Each word in the error message sequence has the following form:

                    pfx          loc, t, count

where:

pfx
    consists of bits 0, 1, and 2, which have the following definitions:

| BIT | DEFINITION |
|---|---|
| 0 | 0 — The sequence continues. |
| | 1 — This is the final parameter in the sequence. |
| 1 | 0 — This is a sequence of an error message. |
| | 1 — This is the entry point to a special editing routine. |
| 2 | 0 — The data is in core storage with the Dump program. |
| | 1 — The data is in the core storage area of the calling program, which may have been overlaid, and may be found on the checkpoint unit. |

loc, t
    is the location of the first word of a sequence of BCD words to be included in the error message. The t may be index registers 1, 2, or 4, loaded by the transfer instruction (TSX) to S.SDMP. If bit 1 of the prefix is 1, loc, t is the entry point of a special editing routine assembled with IBDMP. The prefix bits 0 and 2, and the count are not interpreted. The format of parameters for the special editing routine may be defined by specifications for that routine. Return is to location IDPSC.

count
    is the number of words starting at location loc, t to be included in the error message. Indirect addressing of parameters is indicated by a −1 in the decrement of a parameter. The location and count are found in the location specified by loc, t. Indirect addressing is not available if pfx bit 1 is 1.

If information, directly or indirectly addressed, is not available because s.scK1 is not provided, a word of asterisks will be substituted.

For a special editing routine to have access to data that may have been overlaid and that may be found on the checkpoint unit, a dummy message parameter of the following form:

                    PON          loc, t, erase

may be used before the following parameter:

                    PTW          loc, t

The Dump program will retrieve the data word at loc, t and will store it in location *erase*. If this word has been overlaid and s.scK1 is not provided, *erase* will be set to zero. The Dump program can differentiate between a message parameter, a word retrieval, and indirect addressing, because the length of a segment

of a message must be fewer than 200 words. Location *erase* may not be within the Nucleus and must be below 77777.

There must be a separate sequence of parameters for the on-line and the listed messages. The type parameters must be specified first, followed by the list parameters. Each of the error-message sequences must be terminated by a parameter with pfx bit 0 equal to 1. If a message is not to be typed or listed, the MZE parameter must be supplied. If a special editing routine has completed a message, the parameter counter must point to an MZE parameter upon return to the Dump program.

### Storage to Be Dumped

A sequence of sets of dump parameters follows the two sequences of message parameters. Each set of dump parameters consists of three words of the following form:

| pfx | S.Sunn, , format+mode |
|---|---|
| PZE | from |
| PZE | to |

where:

pfx
    has the following interpretation:

| | | |
|---|---|---|
| Bit 0 | 0 — The sequence continues. | |
| | 1 — This is the final set of parameters in the sequence. | |
| Bit 2 | 0 — The parameters are in core storage with IBDMP. | |
| | 1 — The parameters are in the core storage area of the calling program, which may have been overlaid, and may be found on the checkpoint unit. This is significant only for indirect addressing of parameters. | |

S.Sunn
    is the function to which the external storage medium is assigned; to dump core storage, this field must be zero.

One of the following codes is used to specify the format:

    1 — Octal (single line)
    2 — BCD (single line)
    3 — Octal with BCD (double line)
    4 — Octal with mnemonics (double line)
    5 — Octal with mnemonics and BCD (double line)

If a code is not specified, an error message is given.

Samples of the formats are shown in the section "The Device Print Program" in the publication *IBM 7040/7044 Operating System (16/32K): Programmer's Guide*, Form C28-6318.

The mode is significant for external storage media only. It indicates the method of reading from the external device as follows:

    8 — BCD
    0 — Binary

The symbols *from* and *to* specify the inclusive limits of the dump.

For external storage media, the sequential file number is in the decrement and the record number is in the address.

Indirect addressing is indicated by a first parameter of the form:

pfx         loc, t, −1

where:

pfx
   is as specified above.
loc, t
   is the location in which the set of parameters may be found.

When indirect addressing is used, the remaining two words are not provided.

If a dump parameter, directly or indirectly addressed, is not available because s.SCK1 is not provided, the message DUMP PARAMETER OVERLAID AND LOST will be listed.

If the parameter list is terminated by an MZE word, the remaining two parameters in the set need not be supplied.

To provide an error sequence that does nothing, at least the following parameters must be supplied:

```
PZE     *+6, errno
MZE     TYPE MESSAGE
MZE     LIST MESSAGE
MZE     DUMP CORE
PZE
PZE
```

### Example of An Error Dump

The following is an example of a call to dump in which the parameter sequence and the information typed and listed is the result of a violation of storage protection.

1. Assume that the core storage locations shown in Figure 5 contain the information indicated

2. Assume that the routine shown in Figure 6 is in core storage.

3. Assume that the subroutine shown in Figure 7 is assembled in the Dump program and is used to edit the message.

4. Assume that the parameter entries shown in Figure 8 have been assembled in the Dump program.

The following message will be typed:

    10505 STO VIOLATN JOB DELETED

The following message will be listed:

    10505 IBJOB SPRING 02160 STO VIOLATN LOCAT
      6564 STA INSTR 062100000033

Following the message will be a page heading:

10505 DUMP OF IBJOB SPRING DATE 3/21/64 JOB.
    START 000001735064 ELAPSED
    TIME 03.50 MIN PAGE 10

A conventional panel print and the full core storage print will follow this message.

24



```
00005    IBCLK      777777765350
00032    IXTSP      00C0020C6565
00033               TRA     IXTPS
00211    S.SDAT     000302010604
00213    S.SCLK     000001735064
00214    S.SCIS     77777752720
00217    S.SCUR     IBJOB
00220    S.SFAZ     SPRING
06564               STA     33
```

Figure 5. Assumed Contents of Specified Core Storage Locations



```
1        8          16
         REM        STORAGE PROTECT TRAP
IXTPS    ICT                    .INHIBIT TRAPS
         SXA        IXTP3,4     SAVE WORKING REGISTER
         LXD        IBTSP,4     LOAD ERROR INDICATOR
         TXL        IXTP2,4,0   IS ERROR INDICATED
         TXH        IXTP2,4,3   YES, IS INSTRUCTION RPM
         LXA        IBTSP,4     YES, LOAD LOCATION + 1 OF RPM
         TXH        IXTP2,4,IP2ND+1  IS RELEASE PERMITTED
         LXA        IXTP3,4     YES, RESTORE WORKING REGISTER
         REM        VIOLATION OF MEMORY PROTECT CONDONED
         MIT        S.XTPS      MAY TRAPS BE RESTORED
         RCT                    YES, RESTORE THEM
         TRA*       IBTSP       *RETURN TO REPRIEVED VIOLATOR
         REM        VIOLATION OF MEMORY PROTECT CONDEMNED
IXTP2    STQ        IXTP3+3     HOLD MQ
         LDQ        IBTSP       SAVE ERROR CONDITION WORD
         STQ        IXTP3+1
         LAC        IBTSP,4     SAVE ERROR INSTRUCTION
         LDQ        1,4
         STQ        IXTP3+2
         LDQ        IXTP3+3     RELOAD MQ
         TSX        S.SDMP,4    **THATS ALL CHARLEY
         PZE        ,,10505
IXTP3    PZE        **
         PZE        **,,**      LOCATION + 1 IN ERROR,,ERROR FLAG
         ***        **          INSTRUCTION IN ERROR
         PZE        **          ERASEABLE
```

Figure 6. Routine Assumed to be in Core Storage



```
1        8          16
         REM        PROCESS STORAGE PROTECT TRAP MESSAGE
         REM        CALLING SEQUENCE FOR THIS ROUTINE
         REM        CANNOT BE OVERLAID BY DUMP RECORDS
IDTPS    LXA        IDCX4,1     LOAD CALLING LINKAGE TO IBDMP
         CAL        3,1         LOAD ERROR CONDITION
         PDX        ,2          HOLD ERROR CODE
         SUB        Q1          COMPUTE ERROR LOCATION
         TSX        IXOCV+1,4   $CONVERT LOCATION TO OCTAL
         SLW        IDTP1+3     HOLD LOCATION
         TXL        **+2,2,4    IS BIT PATTERN VALID
         AXT        0,4         NO, SET TO INVALID MESSAGE
         CAL        ICTP1,4     LOAD TRAP TYPE
         SLW        ICTP1+1
         CAL        4,1         LOAD INSTRUCTION IN ERROR
         TSL        IDICV       $CONVERT INSTRUCTION TO OCTAL
         SLW        IDTP1+6     HOLD LEFT HALF
         STQ        IDTP1+7     HOLD RIGHT HALF
         CAL        4,1         LOAD INSTRUCTION IN ERROR
         TSL        IDMCV       $CONVERT INSTRUCTION TO MNEMONIC
         SLW        ICTP1+4     HOLD MNEMONIC
         TRA        IDPSC       *RETURN TO PARAMETER SCAN
         BCI        1,STO VI
         BCI        1,ILL VI
         BCI        1,RPM VI
IDTP1    BCI        1,ILL VI
         BCI        7,     OLATN LOCAT          INSTR
```

Figure 7. Subroutine Assumed to be in the Dump Program



```
1        8          16
         PZE        **+7,,10505      POINTER
         PTW        ICTPS            EDIT ROUTINE
         MZE        ICTP1+1,,2       TYPE SEQUENCE
         MZE        ICTP1+1,,8       LIST SEQUENCE
         MZE        0,,5             DUMP CORE STORAGE
         PZE        0
         PZE        -1
```

Figure 8. Parameter Entries

## Snapshot

A snapshot routine can be used to dump the console and selected areas of core storage. This facility is useful for program debugging.

Any number of snapshots can be taken during the execution of an object program. The following calling sequence is used to request snapshots:

```
TSX          S.SNAP,4
PZE          list,,header
```

where:

list
　　is the location of the first word of a list of parameters that specify the areas of storage for which snapshots are required. This list has the form:

```
list         PZE          fword,,count
             PZE          fword,,count
              .            .
              .            .
              .            .
             PZE          fword,,count
             MZE          fword,,count
```

　　*fword* is the location of the first word of the snapshot area, and *count* is the number of words that are to be preserved. The prefix MZE indicates the end of the list.

header
　　is the location of a BCD word that is to be used in the page heading when the snapshot is printed.

When a request for snapshots is made, the Snapshot routine waits until all channel activity has been completed. It then dumps the contents of the console (the accumulator, MQ, accumulator overflow and divide check indicators, sense switches, entry keys, and index registers) and the specified areas of core storage (in binary) on s.SCK1. When the execution of the object program has been completed, the System Dump program places this binary information with mnemonics and page headings in the format of an octal core storage dump and then writes it on s.SOU1.

If there is no unit corresponding to s.SCK1, a request for snapshots is ignored. If the program is taking checkpoints on s.SCK1, it may not take snapshots. If the program is taking snapshots, it may not take checkpoints on s.SCK1.

## Execution of the Dump Program

Execution of the Dump program involves execution of the Dump routine in the Nucleus before control is passed to the four major phases of the Dump program itself. The core storage used by the Dump program is shown in Figure 9.

### Dump Routine (S.SDMP)

The Dump Routine in the Nucleus saves the console panel, copies the contents of the second and third quarters of the first 16K core storage locations onto the system checkpoint unit, and moves the first quarter of core storage to the third quarter. If a checkpoint unit is not attached, parts of the copied information are lost. The Dump routine uses the System Loader to load the first phase of the Dump program into the upper portion of this core storage area and transfers control to it.

### Phase 1 (IBDMP1)

The first phase of the Dump program consists of a brief routine to copy the contents of the third and fourth quarters of the first of 16K core storage locations on the checkpoint unit to provide working space for Phase 2. Upon completion of this function, it returns control to the System Loader to load IOOP2, IOLS, and the second phase. If the system checkpoint unit is not attached, this routine acts as a link to the second phase.



Figure 9. Use of Core Storage by the Dump Program

## Phase 2 (IBDMP2)

The second phase of the Dump program saves the calling sequence and error number, scans the Error Number Table in this phase, and pieces together the messages to be typed and/or written on the system output unit.

It is possible that a parameter will refer to data in a location that has been overlaid by the Dump program. Phase 2 processes all parameters that refer to data in its own phase, or in that portion of the caller's record that has not been overlaid. It then reads in the portion of core storage that has been saved on s.SCK1 and completes the messages with data from that record.

This phase also sets up a communication region containing information supplied by the dump parameters for use by Phase 3 and, if applicable, writes the console and system panels on the system output unit. It calls the System Loader, which brings in the third phase of the Dump program.

## Phase 3 (IBDMP3)

Using the information in the communication region, this phase writes on the system output unit the areas of core storage and external storage specified in the dump parameters.

*Traceback:* This feature of the Dump program permits the path taken by the program to be followed backward from the location of the calling sequence to the Dump program by using the linkage director set up by the SAVE macro-instruction to determine which subroutines were most recently entered. Each linkage is counted as one level of traceback. The level of traceback is limited to prevent an unending loop of traceback attempts.

*Snapshot Development:* Whenever a programmer uses the Snapshot subroutine, a switch is set to indicate to the System Monitor that snapshots have been taken. Upon return of control to the System Monitor, the Supervisor calls the Dump program to interpret or develop the binary snapshot(s) on the system checkpoint unit. The Dump program rewinds the system checkpoint unit and writes the snapshots on the system output unit in the octal dump format.

*Return Facilities:* These facilities are provided to permit the continuation of the calling program after a dump has been taken. Core storage is restored and control is passed to the final portion of the Restart routine.

If a return is not specified after writing out the specified areas of storage, the Dump program returns control to the System Monitor, which continues to read cards.

## Dump Assembly Option

The following symbol definition appears in the symbolic input to the Dump program, PHASE 2 (IBDMP2):

    IDPDT    SET    n=1

The coding to process the following features of the Dump program will be deleted if n is 0:

1. Traceback
2. System panel dump
3. Dump of I/O device
4. Listed error message text

The deletion of these features shortens the Dump program and allows a full core dump when storage has been memory protected up to 8K.

## Appendix A.  IBJOB Deck Setup Using the Debugging Package (Figure 9)



Figure 10. Example of IBJOB Run Using Debugging Package

## Appendix B.  System Restrictions with Debug Use

1. Checkpoint and restart, or SNAP, may not be used with debugging if the Debug Work Unit is the system checkpoint tape.

2. Location 2, used for linkage to the object-time debugging routines, should not be destroyed by the object program.

3. The deck name of the debugging deck is /DEBUG. This name should not be used for deck or control section names in the job to be debugged.

4. The /DEBUG deck will not stack debug requests. Thus, subroutines called from a debug request may not execute another debug request. This also applies to select and special routines.

5. Real (floating-point) numbers may not be used in arithmetic operations if the user's computer is not equipped with the single-precision floating-point instruction set. In addition, the Debug postprocessor must be reassembled in order to bypass conversion of any values to floating-point, double-precision, or complex modes when listing the debugging dumps (see the publication *IBM 7040/7044 Operating System (16/ 32K): Systems Programmer's Guide*, Form C28-6339).

6. A FORTRAN statement number may not be used as a debugging request point when it is associated with a CONTINUE statement that terminates a DO range when:

    a. the DO increment (DO 1000 I=1, 10, increment) is a variable, or

    b. within a subprogram with adjustable dimensions, the adjustable area is subscripted within the DO range, and the subscript includes the DO variable (I in the example above).

7. Symbolic debugging cannot be used on decks loaded from the subroutine library, nor may a library name appear in the variable field of a $LINK card.

8. The /DEBUG deck will contain virtual references to the real sections that contain symbols referred to in the debug requests. These virtual references are generated prior to processing of $NAME, $USE, and $OMIT cards. As a result, even though $NAME cards may be used to retain at object time two or more control sections of the same name, debugging references may be made to symbols in only *one* of these control sections. If these debugging statements refer to the symbols in the control section that has not been renamed, the user need make no further changes. However, if the control section being referred to is one which has been renamed, the user must provide a $NAME card that includes the deck name /DEBUG as a qualifier.

For example, the following cards will change the control section named A to B in DECK1 and will permit

debugging references to be made to the control section or to symbols within the control section:

```
$NAME      DECK1(A)=B
$NAME      /DEBUG(A)=B
```

9. Under the following circumstances, deck names will be changed to names having the form /nnxyz, where *nn* are decimal digits and *xyz* are the three high-order characters of the original deck name:

a. if debugging is specified for a RELMOD deck whose deck name matches one of the real sections created by an ENTRY pseudo-operation (i.e., has length of zero) and the relative origin of the real section is not zero, or

b. if the original deck name is of the form /nnxyz.

If the deck name appears on a $USE, $OMIT, or $NAME card, however, it will not be changed, and, therefore, debugging references to symbols within this deck may be undefined.

10. =Rn may not refer to ABSMOD decks.

11. Debugging may not be used in an edit run.

12. It is illegal to use symbolic debugging for a job using TCD, unless the debug requests refer to the link immediately preceding the $ENTRY card or unless they refer to the parts of previous links that are not overlaid. No debugging action will occur until the last core load has been loaded. When the last core load has been loaded, debugging STR's will be placed at the locations of all specified request points, whether or not they have been overlaid.

13. Except for the information in the IBNUC Control Dictionary, symbolic information about either ABSMOD decks or absolute text in a RELMOD assembly will not be used by the postprocessor when debugging dumps are listed. Therefore, a debugging dump of an area assembled under an absolute origin will never include its symbol name. In addition, a debugging dump of an instruction that refers to this area will not include as a part of the variable field the symbol name of the area. Modal information will be used to determine the format of a debugging dump of a portion of core storage that has been assembled under an absolute origin only if the element in the LIST or DUMP statement is one of the following: symbol, symbol (subscripts), or (loc1, loc2, mode). Otherwise the mode will be considered octal.

14. $IEDIT cards may not be placed following the debugging packet of a job run. $IEDIT cards may be located preceding the debugging packet, but the occurrence of a $IBDBL card will terminate $IEDIT control and return to s.SIN1 for input.

15. The COPY feature cannot be used in a run which includes load-time symbolic debugging. If COPY is specified, it will be ignored.

16. The user cannot specify DEBUG requests in storage-protected areas, although he can dump portions of storage-protected areas.

17. If the system checkpoint unit is used as the Debug Work Unit, it cannot be assigned to the same physical unit (logical unit for disk or drum) as the unit for the load file.

# Index

A page number in *italics* indicates that the designated reference is of particular importance.